ts.json time series file format
Version 1.0

Table of contents

Format general description

Some of our customers prefer to parse ASCII-like formats for time series, for ease or human reading.

For this reason EMpower will enable exporting to JSON format, which is an ASCII-based format, with the advantage that most modern languages will have already a library to parse JSON files, making the implementation of a reader for these files easy, since no special parsers have to be programmed in most cases. Please consult the list of languages and libraries at https://www.json.org/.

File names

The name of time series files exported to JSON format will have the following structure:

`AAAAA_YYYY-MM-DD-hhmmss_[X].ts.json`

Where "`AAAAA_YYYY-MM-DD-hhmmss`" is the ***recording id***, and is composed of 2 strings separated by underscores (_), the strings represent, in order:

- *`AAAAA`*: Receiver serial number
- *`YYYY-MM-DD-hhmmss`*: Recording start date and time (GPS time, 24 hour fromat)

`[X]`: Is a multi-character field representing the sampling rate of the data contained by the file (i.e. "24000" would mean time series contained by the file has be (re-)sampled at 24000 samples per second).

Therefore, as an exmaple, a time series exported recorded by receiver 45723, started at (GPS time) January 2, 2019, 3:00.PM, and containing the segmented stream sampled at 24KSps, would have the name:

`45723_2019-01-02-150000_24000.ts.json`

Field descriptions

Main header

- **manufacturer**: Should read "Phoenix Geophysics"
- **file_type**: For decimated, segmented time series should read "timeseries_segmented"
- **file_version**: Version for this type of file
- **empower_version**: The string describing the version of EMpower used to generate this file.
- ***recording_id***: Composed of 2 strings separated by underscores (_), the strings represent, in order:
    - Receiver serial number
    - Recording start date and time (YYYY-MM-DD-hhmmss, GPS time)
- **instrument_type**: The commercial name of the receiver used to acquire this time series (for instance MTU-8A or MTU-5C).
- **coords**: Latitude and longitude, in Decimal Degrees format.

- **data_units**: The units for the data values (i.e. “V’ means volts, “AD” means AD raw numbers.In the initial release only volts will be exported).
- **sensor_serials**: Serial numbers of the sensors reportedly connected to the receiver while recording this time series.
- **dipole_lengths_m**: The lengths, in metres, reported for each dipole connected to an electric channel in this time series.

Section “data”

Is an array of blocks of data represented as JSON objects. Each object, will then specify the sampling frequency of the block of data and the timestamp corresponding to the time at which the first sample (for each channel) was recorded.
The block of data will contain data for several channels. The data for each channel is represented in a JSON array, whose id is the channel ID (i.e. “E1”), and the content is a sequence of numbers (i.e. values) in the units specified by the header.

Data block header fields

- **sampling_freq**: Sampling rate for the data contained in this block
- **time_stamp**: Time corresponding to the time at which the first sample of the block (for each channel) was sampled, represented as number of seconds since Jan 01 1970. (GPS time).

Data values

Values packed in arrays for each channel can be written in scientific notation to allow for best precision using a minimum amount of characters. Fixed notation might be used when it makes sense.

Example

```
{
  "manufacturer": "Phoenix Geophysics",
  "file_type": "timeseries_segmented",
  "file_version": "1.0",
  "empower_version": "1.27.0.1:1.27.0.3",
  "recording_id": "10130_2017-10-19-221644",
  "instrument_type": "MTU-5C",
```

```
  "coords": "41.40338, 2.17403",
  "data_units": "V",
  "sampling_freq": 24000,
  "sensor_serials": {
    "H1": "12345",
    "H2": "23456",
    "H3": "34567"
  },
  "dipole_lengths_m": {
    "E1": 50.0,
    "E2": 52.5
  },
  "data": [
    {
      "E1": [0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1],
      "E2": [1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1],
      "H1": [0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1],
      "H2": [1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0],
      "time_stamp": 1234567
    },
    {
      "E1": [0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1],
      "E2": [1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1],
      "H1": [0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1],
      "H2": [1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0],
      "time_stamp": 1234597
    }
  ]
}
```


Annex A. Parsing the file

A large JSON file might not appear easy to parse using common libraries, but there might be libraries designed to parse JSON as a stream (as per this link), or alternatively, you can parse the headers manually, and then pass the internal “data” vectors through a library, allowing the programmer to implement a streamed reader by partially using existing libraries.

Note that although this is not conventional JSON, we have made the data arrays, for instance, in the example above:

```
"E1": [0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1],
```

Be a single line in the file. This means that from the tag describing the channel “E1” to the closing bracket ‘]’ and the following coma, there will not be a carriage return character. This will make it easy to pass these objects to a JSON library from languages that can retrieve files line by line, making it easier for a programmer to create a streamed reader instead of a monolithic all-at-once JSON parser.

Also, the curly braces of the object containing the data vector are in their own line (the closing bracket may only be followed by a comma when necessary). In this way, the streamed reader can scan for opening or closing braces to separate objects to be parsed.

NOTE though, that this format is only guaranteed right after exporting and in a local filesystem that does not reformat ASCII files (i.e. that does not try to append new lines or carriage returns for long lines). Also, transfers over the internet may interpret and reformat the JSON file to another valid representation of the same stream, so if you need to transfer the file via e-mail or network protocols, we recommend compressing it first to ensure that the JSON file is not re-formatted in transit